# One Useful Logic
# That Defines Its Own Truth

Andreas Blass[1] and Yuri Gurevich[2]

[1] Math Dept, University of Michigan, Ann Arbor, MI 48109, USA
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA

**Abstract.** Existential fixed point logic (EFPL) is a natural fit for some applications, and the purpose of this talk is to attract attention to EFPL. The logic is also interesting in its own right as it has attractive properties. One of those properties is rather unusual: truth of formulas can be defined (given appropriate syntactic apparatus) in the logic. We mentioned that property elsewhere, and we use this opportunity to provide the proof.

> *Believe those who are seeking the truth. Doubt those who find it.*
>
> *—André Gide*

## 1   Introduction

First-order logic lacks induction but first-order formulas can be used to define the steps of an induction. Consider a first-order (also called elementary) formula $\varphi(P, x_1, \ldots, x_j)$ where a $j$-ary relation $P$ has only positive occurrences. The formula may contain additional individual variables, relation symbols, and function symbols. In every structure whose vocabulary is that of $\varphi$ minus the symbol $P$ and where the additional individual variables are assigned particular values, we have an operator

$$\Gamma(P) = \{\bar{x} : \ \varphi(P, \bar{x})\}.$$

A relation $P$ is a *closed point* of $\Gamma$ if $\Gamma(P) \subseteq P$, and $P$ is a *fixed point* of $\Gamma$ if $\Gamma(P) = P$. Since $P$ has only positive occurrences in $\varphi(P, \bar{x})$, the operator is monotone: if $P \subseteq Q$ then $\Gamma(P) \subseteq \Gamma(Q)$. By the Knaster-Tarski Theorem, $\Gamma$ has a least fixed point $P^*$ which is also the least closed point of $\Gamma$ [20].

There is a standard way to construct $P^*$ from the empty set by iterating the operator $\Gamma$. Let $P^0 = \emptyset$, $P^{\alpha+1} = \Gamma(P^\alpha)$ and $P^\lambda = \bigcup_{\alpha < \lambda} P^\alpha$ if $\lambda$ is a limit ordinal. There is an ordinal $\alpha$ such that $P^\alpha = P^{\alpha+1} = P^*$. The least such ordinal $\alpha$ is the *closure ordinal* of the iteration. Such elementary inductions have been extensively studied in logic [17, 1].

Notice that we have not really used the fact that $\varphi(P, \bar{x})$ is first-order. One property of $\varphi(P, \bar{x})$ that we used was that $\varphi(P, \bar{x})$ is monotone in $P$, that is that, in every structure of the appropriate vocabulary with fixed values for the additional individual variables, $\Gamma$ is a monotone operator. $\varphi(P, \bar{x})$ could be e.g. a second-order formula monotone in $P$.

The least fixed point $P^*$ can be denoted $\mathrm{LFP}_{P,\bar{x}}\varphi(P,\bar{x})$ and viewed as a $j$-ary relation, so that $[\mathrm{LFP}_{P,\bar{x}}\varphi(P,\bar{x})](y_1,\ldots,y_j)$ functions semantically as a formula. This observation gives rise to an idea to use LFP as a new formula constructor, in addition to propositional connectives and quantifiers. Aho and Ullman [2] indeed suggested to enrich first-order logic with the LFP constructor. The new logic became known as FOL+LFP.

Model checking is polynomial time for any FOL+LFP formula $\psi$. In other words, it can be checked in time polynomial in the size of a finite structure $X$ of the vocabulary of $\psi$ whether $X$ with some values for the free individual variables of $\psi$ is a model of $\psi$. Immerman [16] and Vardi [21] proved that, over ordered finite structures, the converse is true: every property that model checks in polynomial time is expressible in FOL+LFP. In that sense, FOL+LFP captures polynomial time.

Existential fixed point logic (EFPL) is essentially an extension of the existential fragment of first-order logic with the LFP construct. It does not have the universal quantifier and lacks means to simulate universal quantification; see the definition of EFPL in the next section. As far as we know, it was first introduced — in a different guise — by Chandra and Harel [10] in the context of database theory where vocabularies are relational, that is, consist of relation symbols and constants and do not have function symbols of positive arity. Chandra and Harel observed that relational EFPL is equi-expressive with Datalog, a popular database query language.

Existential fixed point logic (EFPL) was further developed by the present authors in [7]; see Section 3. The motivation came from program verification. We noticed that EFPL was appropriate for formulating pre- and post-conditions in Hoare's logic of asserted programs [15]. In particular, the heavy expressivity hypothesis needed for Cook's completeness theorem [12] in the context of first-order logic is automatically satisfied in the context of EFPL.

More recent developments include a deductive system for EFPL introduced by Compton [11] and a normal form for EFPL formulas discovered by Grohe [13], who also studied connections between EFPL and other logics. One of the present authors found connections with topos theory and showed that these connections imply some of the other, previously known, nice properties of EFPL [6, 5]. The other of the present authors, together with Neeman, applied a logic equivalent to EFPL, called liberal Datalog, to develop a powerful authorization language [14]; the equivalence between liberal Datalog and EFPL is shown in detail in [9].

In this note, we recall the definition and known properties of EFPL, and then we prove that the truth definition of EFPL formulas can be given in EFPL.

**Remark 1** Nikolaj Bjørner [4] observed that writing a truth definition for EFPL in EFPL is related to writing an interpreter for EFPL in EFPL. Indeed. But the interesting issue is out of scope here, in this paper, and will have to be addressed elsewhere.

## 2 Existential fixed-point logic: Definition

As indicated in the introduction, existential fixed-point logic differs from first-order logic in two respects, the absence of the universal quantifier and the presence of the least-fixed-point operator. Both of these deserve some clarification.

First we define existential logic EL. Notice that mere removal of the universal quantifier $\forall$ has no real effect on first-order logic, since $\forall x\, \varphi$ can be expressed as $\neg\exists x\, \neg\varphi$. To correctly define the existential fragment of first-order logic, one must prevent such surreptitious reintroduction of the universal quantifier. A traditional way to do that is to insist that all formulas have the prenex existential form $\exists x_1 \ldots \exists x_n \varphi(x_1, \ldots, x_n)$ where $\varphi$ is quantifier-free.

But there is an alternative and more convenient form of the existential fragment proposed in [7]: Allow as propositional connectives only conjunction, disjunction, and negation; use only the existential quantifier; and apply negation only to atomic formulas. It is easy to see that every formula in this alternative fragment is equivalent to one in prenex existential form, and the other way round.

With an eye on the forthcoming introduction of recursion, we stipulate that all relation symbols are divided into two categories: *negatable* and *positive*. And we restrict further the use of negation in the alternative existential fragment of first-order logic: negation can be applied only to atomic formulas with negatable relation symbols. The resulting fragment of first-order logic will be called *existential logic* and denoted EL.

Now we extend existential logic by adding a new formula constructor. As usual, formulas are built by induction from atomic formulas by means of formula constructors. In the case of EFPL, the formula constructors are those of existential logic — the three propositional connectives and the existential quantifier — and one additional LET-THEN constructor that is used to construct induction assertions. We explain how the new constructor works.

Let $\mathcal{F}$ be the collection of formulas constructed so far. A *logic rule* has the form $P(x_1, \ldots, x_j) \leftarrow \delta(P, x_1, \ldots, x_j)$ where $P$ is a positive relation symbol of arity $j$, the $x_i$'s are distinct variables and $\delta$ is any formula in $\mathcal{F}$. We wrote $\delta$ as $\delta(P, x_1, \ldots, x_j)$ to emphasize that it is allowed to contain the relation symbol $P$ and the individual variables $x_1, \ldots, x_j$, but it may also contain additional individual variables, relation symbols, and function symbols. $P$ is the *head symbol* of the rule and $\delta$ is its *body*. Note that the arrow $\leftarrow$ in a logic rule is not the (reverse) implication connective but a special symbol whose only use, in our syntax, is in forming logic rules. A *logic program* is a finite collection of logic rules. (To write a program as text, one needs to order its rules, but the choice of ordering will never matter.) To be compatible with [7], we require that different rules have different head symbols; we could remove this restriction. If $\Pi$ is a program and $\varphi$ is a formula in $\mathcal{F}$ then

$$\text{LET } \Pi \text{ THEN } \varphi$$

is an EFPL formula, an *induction assertion*. If $P(x_1, \ldots, x_j) \leftarrow \delta$ is a rule in $\Pi$ then all occurrences of the variables $x_1, \ldots, x_j$ in the rule are bound occurrences

in the induction assertion. And $P$ is a bound relation variable in the induction assertion.

In general, an occurrence of an individual variable $v$ in a formula $\psi$ is bound if it belongs to a subformula of the form $\exists v \, \alpha$ or to a rule of the form $P(\ldots, v, \ldots) \leftarrow \delta$; otherwise the occurrence is free. The free individual variables of $\psi$ are those with free occurrences in $\psi$. An occurrence of relation symbol $P$ in $\psi$ is bound if it belongs to subformula LET $\Pi$ THEN $\varphi$ of $\psi$ and $P$ is a head symbol of $\Pi$; otherwise the occurrence is free. The vocabulary of $\psi$ consists of all the function symbols in $\psi$ and all relation symbols with free occurrences in $\psi$.

It remains to define the semantics of the induction assertion $\psi = $ LET $\Pi$ THEN $\varphi$. To simplify the exposition, we presume that the program $\Pi$ consists of two rules, $P(x_1, \ldots, x_j) \leftarrow \alpha$ and $Q(y_1, \ldots, y_k) \leftarrow \beta$. In every structure of the vocabulary of $\psi$ with fixed values for the free individual variables of $\psi$, the program gives rise to an operator

$$\Gamma(P, Q) \leftarrow (\{\bar{x} : \ \alpha\}, \{\bar{y} : \ \beta\}).$$

Since $P$ and $Q$ are positive relation symbols, $\Gamma$ is monotone and thus has a least fixed point $(P^*, Q^*)$. To evaluate $\psi$, evaluate $\varphi$ using $P^*$ and $Q^*$ as the values of relations $P$ and $Q$.

## 3    EFPL: Some properties

We describe some properties of EFPL. The default reference is [7].

### Capturing polynomial time

EFPL captures polynomial time computability over structures of the form $\{0, 1, \ldots, n\}$ with (at least) the successor relation and names for the endpoints. In contrast to the corresponding result for FOL+LFP mentioned above, we use the successor relation here rather than the ordering relation $<$. In fact, both proofs depend on the successor relation rather than the order, but in FOL one can define successor in terms of order (but not vice versa), whereas in EFPL one can define order in terms of successor (but not vice versa).

### Validity is r.e. complete

The set of logically valid EFPL formulas is recursively enumerable (in short r.e.). Furthermore, every r.e. set reduces, by means of a recursive function, to the set of valid EFPL formulas. Thus the set of valid EFPL formulas is a complete r.e. set.

### Satisfiability is r.e. complete

The set of satisfiable EFPL formulas is a complete r.e. set.

### Finite validity is co-r.e. complete

The set of EFPL formulas that hold in all finite structures is a complete co-r.e. set. In other words, the set of EFPL formulas $\psi$ such that $\psi$ fails in some finite structure is a complete r.e. set.

### Finite model property

When an EFPL formula $\psi$ is satisfied in a structure $X$, this fact depends on only a finite part of the structure $X$. More precisely, there is a finite subset $D$ of the elements of $X$ such that $\psi$ is satisfied in every structure $X'$ of the vocabulary of $X$ that coincides with $X$ on $D$. Note that $X'$ can be always chosen to be finite. If we allow basic functions of a structure to be partial, then the property in question can be formulated in a particularly simple way: If an EFPL formula is satisfied in a structure then it is satisfied in a finite substructure.

### No transfinite induction is needed

The closure ordinal of any monotone induction

$$P \mapsto \{\bar{x} : \varphi(P, \bar{x})\},$$

where $\varphi$ is EFPL is at most $\omega$, the first infinite ordinal. The definition of the closure ordinal generalizes in a straightforward way to simultaneous monotone induction. The closure ordinal of the induction given by any logic program is at most $\omega$.

### Truth is preserved by homomorphisms

Truth of EFPL formulas is preserved by homomorphisms. Here a homomorphism is a function $h$ from one structure to another such that

- $h$ commutes with (the interpretations of) function symbols,
- $P(a_1, \ldots, a_j)$ implies $P(ha_1, \ldots, ha_j)$
  for every positive relation symbol $P$ of any arity $j$, and
- $P(a_1, \ldots, a_j)$ if and only if $P(ha_1, \ldots, ha_j)$
  for every negatable relation symbol $P$ of any arity $j$.

### EFPL ∩ FOL ⊆ EL

If an EFPL formula $\varphi$ is expressible in first-order logic then $\varphi$ is equivalent to an existential formula. Only a limited form of this result survives in finite model theory. If an EFPL formula $\varphi$ without function symbols and without negations is equivalent, on finite structures, to a first-order formula, then $\varphi$ is equivalent, on finite structures, to an existential formula without negations [3, 18]. This fails even if $\varphi$ has no function symbols and only the equality relation is negatable [3, Section 10].

# 4   Prerequisites for truth

Our objective in the rest of the article is to show that EFPL can formalize its own truth definition. That is, we shall define, in EFPL with suitable vocabulary, truth of EFPL sentences (that is formulas with no free variables) of the same vocabulary. We use the term predicate to mean a relation symbol or a relation depending on the context.

Since sentences are built from subformulas that may have free variables, we shall actually define the slightly more general concept of satisfaction of formulas by assignments of values to the free variables. The need to define the more general notion of satisfaction of formulas in order to obtain truth for sentences is familiar from first-order logic.[3] A new complication, of the same general nature, arises in EFPL. The bound predicates of a sentence $\varphi$ are free in some subformulas of $\varphi$. We should define satisfaction of $\varphi$ in a structure whose vocabulary does not include those predicates. But the definition will pass through subformulas of $\varphi$ whose satisfaction will depend on the interpretations of those predicates. As a result, we need to define satisfaction of $\varphi$ in a context that includes not only a structure (for the vocabulary of $\varphi$) and an assignment of values to the free variables of $\varphi$ (as in FOL) but also the logic rules that provide the meaning of all other predicates that occur in $\varphi$ — or that occur in the bodies of those rules.

Let $\varUpsilon$ be a vocabulary and $X$ a structure of vocabulary $\varUpsilon$. Any predicate that does not occur in $\varUpsilon$ will be called an *extra predicate*. We shall define satisfaction in $X$ for $\varUpsilon$-formulas. Requirements will be imposed shortly on $\varUpsilon$ and $X$, but for now $\varUpsilon$ is just some vocabulary and $X$ some $\varUpsilon$-structure. We intend to define, in EFPL, a ternary predicate Sat such that, when

- the value of its first argument is a formula $\varphi$, of vocabulary $\varUpsilon$ plus (possibly) some extra predicates,
- the value of its second argument is a logic program $\varPi$ whose head predicates include all extra predicates that occur in $\varphi$ or $\varPi$, and
- the value of its third argument is an assignment $s$ of elements of $X$ to (at least) all individual variables that are free in $\varphi$ or in $\varPi$,

then the truth value of $\mathrm{Sat}(\varphi, \varPi, s)$ in $X$ is the same as the truth value, in $X$, of $\varphi$ with values for its variables given by $s$ and with the extra predicates interpreted by the least fixed point of (the monotone operator defined by) $\varPi$.

Furthermore, we do not intend to use any clever tricks in our definition of Sat. It will be a formalization of the explanation given above (and in [7]) of the meaning of EFPL formulas. The point of this work is to show that this formalization can be carried out in EFPL itself.

---

[3] A few authors, notably Shoenfield [19], define truth directly. To do so, they expand the vocabulary by adding constants for all elements of the structure under consideration, and instead of assigning values to variables they substitute constants for variables. We could have used this approach for EFPL, but we chose to parallel the more widely used approach in FOL, via satisfaction.

For all this to make sense, the structure $X$ must contain the formulas $\varphi$ of EFPL, the logic programs $\Pi$, and the assignments $s$. Furthermore, the vocabulary must be adequate to express the basic syntactic properties of formulas and to allow basic constructions of assignments, rules, and programs. We do not, however, wish to specify the exact syntactic nature of formulas — for example, are they sequences of symbols, or are they parse trees, or are they Gödel numbers? Our work is independent of such details. So we shall merely assume that certain notions (e.g., the operation of forming the conjunction of two formulas) are expressible; the details of how they are expressed (and which notions are primitive and which are derived) are irrelevant.[4]

In the rest of this section, we list what we require of our vocabulary $\Upsilon$ and structure $X$, occasionally adding some comments about the reasons for particular requirements.

$\Upsilon$ should be finite. The reason is that the definition of satisfaction must, in the clauses for atomic formulas, use all the relation and function symbols of $\Upsilon$.

The equality predicate should be negatable. The reason is that the notion of EFPL formula requires some things to be distinct, for example the variables in the head of a rule and the head symbols of different rules in a program.

$X$ should contain a copy $\mathbb{N}$ of the natural numbers, and $\Upsilon$ should have a constant symbol for $0$ and a unary function symbol $S$ for successor. $\mathbb{N}$ itself, as a unary relation, is definable:

$$\mathbb{N}(x) :\equiv \text{LET } N(z) \leftarrow z = 0 \ \lor \ \exists y \, (N(y) \land z = S(y)) \text{ THEN } N(x).$$

We could also define addition and multiplication as ternary relations, and the ordering, and similarly for other primitive recursive functions and relations.

We need $\mathbb{N}$ primarily to index elements of lists, for example the list of terms that serves as the arguments of a relation or function symbol. Since $\Upsilon$ is finite, we could handle the argument lists of its own relation and function symbols in an ad hoc manner, without a general notion of natural number or of list. But EFPL imposes no bound on the arities of the head symbols of logic rules, so atomic formulas can involve arbitrarily long argument lists, and natural numbers are needed for treating these.

Although EFPL does not allow universal quantification in general, it can simulate universal quantification over finite initial segments of $\mathbb{N}$, as shown by the following lemma from [7].

**Lemma 2** *For any EFPL formula $\varphi(x)$, there is an EFPL formula $\psi(y)$ equivalent, for all $y \in \mathbb{N}$, to $(\forall x < y) \, \varphi(x)$.*

*Proof.* The most natural choice of $\psi(y)$ describes a search from $0$ up to $y$:

LET $K(x) \leftarrow x = 0 \ \lor \ \exists w \, \big(x = S(w) \ \land \ K(w) \ \land \ \varphi(w)\big)$ THEN $K(y)$. $\qquad \square$

---

[4] We shall occasionally indicate how certain notions can be defined from others in EFPL. Those indications can help to reduce the assumptions needed about $\Upsilon$.

**Convention 3** Consider the definition of $\mathbb{N}$ exhibited above, and notice that its essential content is contained in the rule

$$N(z) \leftarrow z = 0 \vee \exists y \, (N(y) \wedge z = S(y)),$$

which makes the bound predicate symbol $N$ denote the set of natural numbers. The rest of the definition,

$$\mathbb{N}(x) :\equiv \text{LET } \ldots \text{ THEN } N(x),$$

merely transfers this denotation to the defined notation $\mathbb{N}$. Instead of introducing a bound predicate variable $N$ to, in effect, duplicate the desired predicate $\mathbb{N}$, we could convey the same information by writing

$$\mathbb{N}(z) :\leftarrow z = 0 \vee \exists y \, (\mathbb{N}(y) \wedge z = S(y)).$$

Although this is not an EFPL formula, we adopt the convention that it is to serve as an abbreviation of the definition of $\mathbb{N}$ displayed earlier. In general, when we write a rule with a colon before the $\leftarrow$, it is to be interpreted as defining a formula. Thus,

$$\mathbb{P}(\bar{x}) :\leftarrow \delta(\mathbb{P}, \bar{x})$$

means that $\mathbb{P}(\bar{x})$ is defined as the formula

$$\text{LET } Q(\bar{z}) \leftarrow \delta(Q, \bar{z}) \text{ THEN } Q(\bar{x}).$$

**Convention 4** Later, we shall also need to deal with definitions of this sort in which the body $\delta$ is a disjunction of many subformulas. For example, our ultimate goal, the definition of Sat, will have several disjuncts, covering the different syntactic constructs of EFPL. In such cases, it is convenient to present one disjunct (or a small number of them) at a time. Thus, for a small example, the definition of $\mathbb{N}$ above could be broken into two parts:

$$\mathbb{N}(z); \leftarrow z = 0$$
$$\mathbb{N}(z); \leftarrow \exists y \, (\mathbb{N}(y) \wedge z = S(y)).$$

We use a semicolon before $\leftarrow$ (instead of a colon) to indicate that the full definition involves more disjuncts. (This use of a semicolon as a partial colon is suggested by the word "semicolon.") In general, if we write several semicolon definitions $\mathbb{P}(\bar{x}); \leftarrow \delta_i$ for the same $P(\bar{x})$, then they are to be understood as meaning $\mathbb{P}(\bar{x}) :\leftarrow \bigvee_i \delta_i$.

Returning to the requirements on $X$ and $\Upsilon$, we require $X$ to contain the variables and the assignments. The latter are finite partial functions from the variables into (the universe of) $X$. $\Upsilon$ should define a predicate Vbl for the set of variables, a constant symbol $\varnothing$ for the empty assignment, and a ternary function symbol Modify for the function defined as follows: Given an assignment $s$, a variable $v$, and an element $a$ of $X$, Modify$(s, v, a)$ is the assignment $t$ that sends $v$ to $a$ and otherwise agrees with $s$ (whether or not $a$ is in the domain of $s$).

**Convention 5** Here and in what follows, we use the terminology "$\Upsilon$ should define a predicate for" some relation on $X$ to mean that there should be an EFPL formula in vocabulary $\Upsilon$ whose truth set in $X$ is the desired relation. Of course, the easiest way to arrange this would be for the given relation to be one of the basic relations of $X$, so that the required EFPL formula would be atomic. But it will never matter whether the formula is atomic or not.

Similarly, when we ask that $\Upsilon$ should have certain function symbols, we could weaken that to require only some terms, possibly involving nesting of function symbols, and our proofs would be unchanged.

We also need to express "$s$ is an assignment," "$v$ is in the domain of $s$," and "$s(v) = a$," but we need not assume these separately, as they are definable from $\varnothing$ and Modify. They are given, using our conventions above and the familiar convention of (existentially) quantifying several variables at once, by

$$\mathrm{Assgt}(s); \leftarrow s = \varnothing$$
$$\mathrm{Assgt}(s); \leftarrow \exists t, v, a\,(\mathrm{Assgt}(t) \wedge \mathrm{Vbl}(v) \wedge s = \mathrm{Modify}(t, v, a))$$
$$v \text{ inDom } s :\leftarrow \exists t, a\,(s = \mathrm{Modify}(t, v, a)).$$
$$s(v) = a :\leftarrow \exists t\,(s = \mathrm{Modify}(t, v, a))$$

Note that here $s(v) = a$ is defined as a ternary relation, not as an instance of equality.

We shall also need to have, among the elements of $X$, the relation and function symbols of $\Upsilon$ as well as the extra predicates available as head symbols of rules. Each relation symbol $P$ or function symbol $f$ of $\Upsilon$, should be denoted by a closed term $\dot{P}$ or $\dot{f}$ of $\Upsilon$. (We remain flexible as to what the symbols of $\Upsilon$ should be. For example, they could be Gödel numbers, and then their names $\dot{P}$ and $\dot{f}$ could be terms of the form $SS\ldots S(0)$. But there are many other options, and all will work. Note, however, that we cannot take all the $\dot{f}$'s to be simple constant symbols, as they would then be among the $f$'s, and there would not be enough room in a finite $\Upsilon$ for all of these names to have names.)

The extra predicates available as head symbols of rules should have specified numbers of arguments. That is, there should be an $\Upsilon$-definable predicate Arity such that $\mathrm{Arity}(a, n)$ holds in $X$ (for elements $a, n \in X$) if and only if $a$ is one of these head predicate symbols and $n \in \mathbb{N}$ is the number of its argument places.

As mentioned earlier, we shall need lists, so we require that $X$ contain all lists (i.e., finite sequences) of elements of $X$. The vocabulary $\Upsilon$ should contain at least the constant Nil, denoting the empty list, and the binary function symbol Append, for the function that lengthens a list by adding one element at the end. Thus, for example,

$$\langle a, b, c \rangle = \mathrm{Append}(\mathrm{Append}(\mathrm{Append}(\mathrm{Nil}, a), b), c).$$

Other predicates and functions that we shall need for dealing with lists can be defined in terms of Nil and Append.

$$\mathrm{List}(l); \leftarrow l = \mathrm{Nil}$$
$$\mathrm{List}(l); \leftarrow \exists x, a \, (\mathrm{List}(x) \wedge l = \mathrm{Append}(x, a))$$
$$l \; \mathrm{hasLength} \; n; \leftarrow l = \mathrm{Nil} \wedge n = 0$$
$$l \; \mathrm{hasLength} \; n; \leftarrow \exists x, a, m \, \big(l = \mathrm{Append}(x, a) \wedge x \; \mathrm{hasLength} \; m \wedge n = S(m)\big)$$
$$(l)_i = a; \leftarrow \exists x \, \big(x \; \mathrm{hasLength} \; i \wedge l = \mathrm{Append}(x, a)\big)$$
$$(l)_i = a; \leftarrow \exists x, b \, \big((x)_i = a \wedge l = \mathrm{Append}(x, b)\big)$$
$$\mathrm{Cat}(a, b, l); \leftarrow b = \mathrm{Nil} \wedge l = a$$
$$\mathrm{Cat}(a, b, l); \leftarrow \exists c, x, m \, \big(\mathrm{Cat}(a, c, m) \wedge$$
$$b = \mathrm{Append}(c, x) \wedge (l = \mathrm{Append}(m, x)\big).$$

Here $(l)_i = a$, though it looks like an equation, is really a defined ternary relation, whose meaning is that $a$ is the $i^{\mathrm{th}}$ component of the list $l$, where we start counting with 0, and where the length of $l$ must be at least $i + 1$ so that there is an $i^{\mathrm{th}}$ term. And "Cat" alludes to "concatenation". If $a, b, l$ are lists and $\mathrm{Cat}(a, b, l)$ holds, then $l$ is the concatenation $a * b$ of $a$ and $b$.

We note the following consequence of Lemma 2, allowing universal quantification over the elements of a list.

**Corollary 6** *For any EFPL formula $\varphi(x)$, there is an EFPL formula $\psi(y)$ that holds, when the value of $y$ is a list, if and only if $\varphi$ holds of all elements of that list. That is, $\psi(y)$ is the result of universally quantifying $\varphi(x)$ over all elements $x$ of the list $y$.*

*Proof.* Use Lemma 2 to express
$$\exists n \, \big(y \; \mathrm{hasLength} \; n \wedge (\forall i < n) \, \exists z \, ((y)_i = z \wedge \varphi(z))\big). \qquad \square$$

It will be convenient to write $(\forall x \in y) \, \varphi(x)$ for the formula $\psi$ given by this corollary.

Finally, $X$ must contain the syntactic entities relevant to EFPL, such as terms, logic rules, logic programs, and formulas. The precise nature of these entities depends on arbitrary choices of how to represent syntax. We require merely that some representation be present and that $\Upsilon$ be able to describe fundamental syntactic relationships.

First, $\Upsilon$ should have a binary function symbol Apply, used to form a compound term $f(t_1, \ldots, t_n)$ from an $n$-ary function symbol $f$ and a list $\langle t_1, \ldots, t_n \rangle$ of $n$ terms, and also used similarly to form atomic formulas $P(t_1, \ldots, t_n)$. Depending on how syntax is represented, Apply could, for example, be simply a pairing function, or it could be the operation of prepending an element to a list, or it could produce a tree from a root and its immediate subtrees, or it could be an arithmetical operation on Gödel numbers.

There should also be a unary function symbol Neg and binary function symbols Conj, Disj, Quant, and IndAsrt for the operations of negating a formula,

forming conjunctions, forming disjunctions, forming existential quantifications, and forming induction assertions LET $\Pi$ THEN $\varphi$. The arguments of these operations are intended to be formulas, except that the first argument of Quant is the variable being quantified and the first argument of IndAsrt is the program that goes between LET and THEN.

There should also be a binary function symbol Rule for the operation building a logic rule from its head and its body. We shall take logic programs to be (certain) lists of rules, so we do not need additional capabilities in $\Upsilon$ to handle these. (We could have used sets of rules instead, but then $\Upsilon$ would need additional capabilities.) Finally, there is a ternary relation RenameAway such that, if $\Pi$ is a program and $\varphi$ is a formula and RenameAway$(\varphi, \Pi, \varphi')$ holds, then $\varphi'$ is a formula obtained from $\varphi$ by renaming the bound predicates of $\varphi$ away from the head predicates of $\Pi$, so that the formula $\varphi'$ is equivalent to $\varphi$, and no head predicate of $\Pi$ is bound in $\varphi'$.

This completes our requirements on $\Upsilon$ and $X$. They can be summarized thus: EFPL syntax and basic combinatorial ingredients for EFPL semantics (like assignments) are available in $X$ and expressible in EFPL in vocabulary $\Upsilon$.

## 5 Semantics of terms

Terms are built, as in FOL, by starting with variables and iteratively applying function symbols. The definition is formalized as follows.

$$\mathrm{Term}(t); \leftarrow \mathrm{Vbl}(t)$$

$$\mathrm{Term}(t); \leftarrow \exists l \big( t = \mathrm{Apply}(\dot{f}, l) \wedge \mathrm{List}(l) \wedge l \text{ hasLength } \hat{n} \wedge (\forall x \in l) \mathrm{Term}(x) \big).$$

Here the second line is to be repeated for each function symbol $f$ of $\Upsilon$, $n$ is the arity of $f$, and $\hat{n}$ is the numeral for $n$, namely $SS \ldots S(0)$ with $n$ occurrences of $S$. Recall that the universal quantification $\forall x \in l$ was introduced after Corollary 6 as an abbreviation of an EFPL formula. Recall also that $\Upsilon$ is finite, so there is no difficulty writing the appropriate line for each $f$.

Semantically, a term gets a value (in the given structure $X$) once an assignment provides values for all the variables in $t$. So the values of terms are given by a binary function, whose arguments are a term and an assignment. To define it recursively, we regard this binary function as a ternary relation, and we define it as follows.

$$\mathrm{Val}(t, s, a); \leftarrow \mathrm{Vbl}(t) \wedge \mathrm{Assgt}(s) \wedge s(t) = a$$

$$\mathrm{Val}(t, s, a); \leftarrow \exists l, u_0, \ldots, u_{n-1}, b_0, \ldots, b_{n-1}$$

$$\big( t = \mathrm{Apply}(\dot{f}, l) \wedge \mathrm{List}(l) \wedge l \text{ hasLength } \hat{n} \wedge \mathrm{Assgt}(s)$$

$$\wedge \bigwedge_{i < n} ((l)_i = u_i \wedge \mathrm{Val}(u_i, s, b_i)) \wedge a = f(b_1, \ldots, b_n) \big).$$

The explanatory comments after the definition of Term apply here as well.

**Remark 7** In principle, we could do without the definition of Term. The definition of Val assigns values only to terms in any case. But it would do no harm if Val were defined in some extraneous cases, as long as it worked correctly for terms.

## 6   Semantics of formulas

As indicated earlier, the semantics of a formula involves not only the structure $X$ and an assignment $s$ but also a collection $\Pi$ of logic rules to determine the meaning of any extra predicates used in the formula but not bound by LET-THEN constructions in the formula. Ultimately, when we deal with $\Upsilon$-formulas, there will be no such extra predicates, so $\Pi$ will be irrelevant, but in the recursive construction of an $\Upsilon$-formula (and in the recursive definition of its satisfaction), subformulas can occur that do use extra predicates. So we shall define Sat as a ternary predicate, where the intended meaning of $\mathrm{Sat}(\varphi, \Pi, s)$ is that the formula $\varphi$ is true, in our given structure $X$, when the extra predicates are interpreted by the least fixed point of $\Pi$ and the free variables are assigned values by $s$.

The definition of Sat will have numerous clauses, according to the last constructor used in building $\varphi$, so we shall make much use of the ";$\leftarrow$" convention. This way, we can present the clauses one (or a few) at a time and insert comments and even other definitions between them.

We begin with the case of atomic formulas whose predicates are from $\Upsilon$. The definition is quite analogous to the earlier definition of the values of terms.

$$\mathrm{Sat}(\varphi, \Pi, s); \leftarrow \exists l, u_0, \ldots, u_{n-1}, b_0, \ldots, b_{n-1}$$
$$\left(\varphi = \mathrm{Apply}(\dot{P}, l) \wedge \mathrm{List}(l) \wedge l \text{ hasLength } \hat{n} \wedge \mathrm{Assgt}(s) \right. \tag{1}$$
$$\left. \wedge \bigwedge_{i<n}((l)_i = u_i \wedge \mathrm{Val}(u_i, s, b_i)) \wedge P(b_1, \ldots, b_n)\right).$$

This is to be repeated for all of the (finitely many) predicates $P$ of $\Upsilon$ with $n$ being the arity of $P$. As before, $\hat{n}$ is the numeral for $n$.

The case of negated atomic formulas is almost the same; of course it is to be repeated only for negatable $P$.

$$\mathrm{Sat}(\varphi, \Pi, s); \leftarrow \exists l, u_0, \ldots, u_{n-1}, b_0, \ldots, b_{n-1}$$
$$\left(\varphi = \mathrm{Neg}(\mathrm{Apply}(\dot{P}, l)) \wedge \mathrm{List}(l) \wedge l \text{ hasLength } \hat{n} \wedge \mathrm{Assgt}(s) \right. \tag{2}$$
$$\left. \wedge \bigwedge_{i<n}((l)_i = u_i \wedge \mathrm{Val}(u_i, s, b_i)) \wedge \neg P(b_1, \ldots, b_n)\right).$$

Rather than continuing with the remaining atomic formulas, those that use extra predicates, let us first dispose of the remaining "easy" clauses, those not involving $\Pi$.

$$\mathrm{Sat}(\varphi, \Pi, s); \leftarrow \exists \alpha, \beta \left(\varphi = \mathrm{Conj}(\alpha, \beta) \wedge \mathrm{Sat}(\alpha, \Pi, s) \wedge \mathrm{Sat}(\beta, \Pi, s)\right)$$
$$\mathrm{Sat}(\varphi, \Pi, s); \leftarrow \exists \alpha, \beta \left(\varphi = \mathrm{Disj}(\alpha, \beta) \wedge (\mathrm{Sat}(\alpha, \Pi, s) \vee \mathrm{Sat}(\beta, \Pi, s)\right) \tag{3}$$
$$\mathrm{Sat}(\varphi, \Pi, s); \leftarrow \exists \alpha, v, a \left(\varphi = \mathrm{Quant}(v, \alpha) \wedge \mathrm{Sat}(\alpha, \Pi, \mathrm{Modify}(s, v, a))\right)$$

This completes the easier part of the definition of Sat, the part concerning just EL. To complete the definition for EFPL, we must deal carefully with programs in both of their roles — as the second argument of Sat and as a constituent of induction assertions.

This will require some preliminaries. First, we need the notion of a list with no repetitions.

$$\text{1-1-List}(l) :\equiv \exists n \left(l \text{ hasLength } n \wedge \right.$$
$$\left. (\forall i, j < n) \, \exists x, y \, ((l)_i = x \, \wedge \, (l)_j = y \wedge (i = j \, \vee \, \neg(x = y)))\right).$$

We also need a construction that amounts to applying a unary function to each element of a list, producing a new list. The situation is complicated by the fact that our unary functions are often given as binary relations. We therefore adopt the following notation. If we have defined a binary relation $R$, then we write $R^+$ for the relation defined as follows.

$$R^+(l, m) :\equiv \exists n \left(l \text{ hasLength } n \, \wedge \, m \text{ hasLength } n \wedge \right.$$
$$\left. (\forall i < n) \, \exists u, v \, ((l)_i = u \, \wedge \, (m)_i = v \, \wedge \, R(u, v))\right).$$

For example, let us define HS (abbreviating "head symbol") by

$$HS(r, p) :\equiv \exists y, z \, (r = Rule(Apply(p, y), z)).$$

Then when $\Pi$ is a list of rules, $HS^+(\Pi, m)$ means that $m$ is the list of their head symbols. One of the requirements for a program is that this list $m$ be one-to-one, so there will be a clause $\exists m \, (HS^+(\Pi, m) \wedge \text{1-1-List}(m))$ in the definition of program.

We shall also use the plus-notation with a parameter. Specifically, we think of $Val(u, s, b)$ as the graph of a function $u \mapsto b$ with $s$ fixed, so the plus-notation makes $Val^+(\bar{u}, s, \bar{b})$ the relation between a list of terms and their values, all for the same assignment $s$. We refrain from writing out the definition, since it's just like the definition of $R^+$ above, with the extra argument $s$ inserted into both $R$ and $R^+$.

We need an improved version of the function Modify, to modify an assignment by mapping all the variables in a list $l$ to the corresponding values in another list $q$ (of the same length).

$$\text{Change}(s, l, q, r); \leftarrow l = \text{Nil} \wedge q = \text{Nil} \wedge s = r$$
$$\text{Change}(s, l, q, r); \leftarrow \exists l', q', r', v, a \left(l = \text{Append}(l', v) \wedge q = \text{Append}(q', a)\right.$$
$$\left. \wedge \text{ Change}(s, l', q', r') \wedge r = \text{Modify}(r', v, a)\right).$$

With these preliminaries, we can write down the definition of satisfaction for atomic formulas that begin with one of the extra predicates. The idea is to find, in $\Pi$, the rule having that symbol as its head symbol, and to use the body of that rule as the criterion of truth for our atomic formula. It will be useful later

to make sure that the $\Pi$ in the second argument place of Sat has no repeated head symbols, so we include that in the definition.

$$\text{Sat}(\varphi, \Pi, s); \leftarrow \exists p, t, k, i, m, l, r, q, \delta$$
$$\big(\varphi = \text{Apply}(p, t) \wedge t \text{ hasLength } k \wedge \text{Arity}(p, k) \wedge$$
$$(\forall x \in t) \, \text{Term}(x) \wedge \text{HS}^+(\Pi, m) \wedge \text{1-1-List}(m) \wedge$$
$$(\Pi)_i = \text{Rule}(\text{Apply}(p, l), \delta) \wedge \text{1-1-List}(l) \wedge \tag{4}$$
$$l \text{ hasLength } k \wedge (\forall x \in l) \, \text{Vbl}(x) \wedge \text{Val}^+(t, s, q) \wedge$$
$$\text{Change}(s, l, q, r) \wedge \text{Sat}(\delta, \Pi, r)\big).$$

In prose, the essential part of this says that $\varphi$ has the form $p(\bar{t})$ for an extra predicate of arity $k$, with $\bar{t}$ being a $k$-tuple of terms; that $\Pi$ contains a rule $p(\bar{l}) \leftarrow \delta$ with head $p$, $\bar{l}$ being a $k$-tuple of distinct variables; and that $\delta$ is satisfied by the assignment $r$ obtained from $s$ by replacing each of the variables in the list $\bar{l}$ by the value of the corresponding element of $\bar{t}$. This replacement amounts, intuitively, to taking the definition of $p(\bar{l})$ as $\delta(\bar{l})$ and applying it to $p(\bar{t})$, the terms $\bar{t}$ replacing the variables $\bar{l}$. Instead of doing a syntactic substitution of $\bar{t}$ for $\bar{l}$ in $\delta$, we have made the corresponding semantic change, assigning to the variables in $\bar{l}$ the values of the terms in $\bar{t}$.

It may seem strange that this clause in the definition of Sat says nothing about iterating the operator defined by $\delta$. After all, $p$ should be interpreted as the least fixed point of that operator. But the desired iteration is automatically accomplished by the iteration involved in the definition of Sat. That is, if $p$ occurs in $\delta$, then the true instances of $p$ can contribute to the true instances of $\delta$ and can thereby contribute to additional true instances of $p$.

We must still provide the clause for induction assertions in our definition of Sat. Fortunately, this is relatively easy, since iteration is already implicitly done in the preceding clause.

$$\text{Sat}(\varphi, \Pi, s) \; ; \leftarrow \; \exists \varphi', \Sigma, \alpha, \Theta$$
$$\big(\text{RenameAway}(\varphi, \Pi, \varphi') \wedge \varphi' = \text{IndAsrt}(\Sigma, \alpha) \tag{5}$$
$$\wedge \; \text{Cat}(\Pi, \Sigma, \Theta) \wedge \text{Sat}(\alpha, \Theta, s)\big).$$

Here $\varphi'$ is equivalent to $\varphi$ and so $\text{Sat}(\varphi, \Pi, s)$ should be equivalent to $\text{Sat}(\varphi', \Pi, s)$. Further, $\varphi' = \text{LET } \Sigma \text{ THEN } \alpha$, and no head predicate of $\Pi$ is bound in $\varphi'$. It follows that the head predicates of $\Pi$ are disjoint from the head predicates of $\Sigma$, so that the concatenation $\Theta$ of $\Pi$ and $\Sigma$ is a legitimate program. Accordingly $\text{Sat}(\varphi', \Pi, s)$ should be equivalent to $\text{Sat}(\alpha, \Theta, s)$.

That concludes the definition of $\text{Sat}(\varphi, \Pi, s)$. It is easy to see that it works as intended. In the case when $\varphi$ is a sentence and when both $\Pi$ and $s$ are empty, $\text{Sat}(\varphi, \Pi, s)$ holds in the structure $X$ if and only $\varphi$ does.

## References

1. Peter Aczel, "An introduction to inductive definitions," in *Handbook of Mathematical Logic*, J. Barwise, editor, North Holland, 1977.

2. Alfred V. Aho and Jeffrey D. Ullman, "Universality of data retrieval languages," 6th ACM Symp. on Principles of Programming Languages (POPL 1979) 110–119.

3. Miklos Ajtai and Yuri Gurevich, "Datalog vs first-order logic," *J. Comput. System Sci.* 49 (1994) 562–588.

4. Nikolaj Bjørner, private communication, June 2008.

5. Andreas Blass, "Topoi and computation," *Bull. Eur. Assoc. Theor. Comput. Sci.* 36 (1988) 57–65.

6. Andreas Blass, "Geometric invariance of existential fixed-point logic," in *Categories in Computer Science and Logic*, J. Gray and A. Scedrov eds., Contemp. Math. 92 (1989) 9–22.

7. Andreas Blass and Yuri Gurevich, "Existential fixed-point logic," in *Computation Theory and Logic,* E. Börger ed., Springer-Verlag, Lecture Notes in Computer Science 270 (1987) 20–36.

8. Andreas Blass and Yuri Gurevich, "The underlying logic of Hoare Logic," *Bull. Eur. Assoc. Theor. Comput. Sci.* 70 (2000) 82–110, reprinted in *Current Trends in Theoretical Computer Science: Entering the 21st Century,* G. Paun, G. Rozenberg and A. Salomaa eds., World Scientific (2001) 409–436.

9. Andreas Blass and Yuri Gurevich, "Two forms of one useful logic: Existential fixed point logic and liberal Datalog" *Bull. Eur. Assoc. Theor. Comput. Sci.* 95, June 2008.

10. Ashok Chandra and David Harel, "Horn clause queries and generalizations," *Journal of Logic Programming* 1 (1985) 1–15.

11. Kevin J. Compton, "A deductive system for existential fixed point logic," *Journal of Logic and Computation* 3 (1993) 197–213.

12. Stephen Cook, "Soundness and completeness of an axiom system for program verification," *SIAM J. Computing* 7 (1978) 70–90.

13. Martin Grohe, "Existential least fixed-point logic and its relatives," *J. Logic Comput.* 7 (1997) 205–228.

14. Yuri Gurevich and Itay Neeman, "DKAL: Distributed knowledge authorization language," in 21st IEEE Computer Security Foundations Symp. (CSF 2008).

15. C. A. R. Hoare, "An axiomatic basis for computer programming," *Comm. ACM* 12 (1969) 576–580, 583.

16. Neil Immerman, "Relational queries computable in polynomial time," 14th ACM Symp. on Theory of Computing (STOC 1982) 147–152.

17. Yiannis N. Moschovakis, *Elementary Induction on Abstract Structures,* Studies in Logic and the Foundations of Mathematics 77, North-Holland (1974).

18. Benjamin Rossman, "Existential positive types and preservation under homomorphisms," 20th Annual IEEE Symp. on Logic in Computer Science (LICS 2005) 467–476.

19. Joseph Shoenfield, *Mathematical Logic,* Addison-Wesley (1967). Reprinted by the Association for Symbolic Logic and A K Peters (2001).

20. Alfred Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics* 5:2 (1955) 285-309.

21. Moshe Vardi, "The complexity of relational query languages," Fourteenth ACM Symp. on Theory of Computing (STOC 1982) 137–146.